Manu Shantharam <shantharam.manu@gmail.com>

code variants derived by algorithm

2 messages

Jacqueline Chame <jacqueline.chame@gmail.com> Wed, Jan 2, 2013 at 10:10 AM
To: snarayan@mcs.anl.gov, shantharam.manu@gmail.com

Krishna and Manu

Here are the versions of matrix multiplication that I derived by hand, according with the algorithm in Chun's CGO'05 paper.

```
==============

Code variants for matrix multiplication, from the initial variant v0 and
assuming that MostProfitableLoops() returns a single loop.

v0 = {
do I = 1, M
  do J = 1, M
    do K = 1, M
      C(I,J) = C(I,J) + A(I,K) * B(K,J)
}


The initial values for sets Loops and Refs are:
Loops = {I, J, K}
Refs = {{C(I,J)read, C(I,J)write}, A(I,K), B(K,J)}

And the initial values of V and level are:
V = {v0}
level = 0

I'm using an additional set, WorkingLoops, in DeriveVariants, and each variant v has
its own set v.WorkingLoops.

============================
First iteration of while loop
============================
MostProfitableLoops() returns loop K, which carries temporal reuse for C(I,J)read and
C(I,J)write:
L = MostProfitableLoops(Loops, Refs)
L = {K}

Vnew = {}

GenVariant is called with parameters v0, K, 0 (GenVariant(v0, K, 0).
```

GenVariant unrolls loops I and J to exploit the reuse of C(I,J) in registers, returning v1.
This is a simplified version of v1 assuming that both unroll factors UI and UJ are equal to 2:

```
v1  = {
do Iu = 1, M, 2
  do Ju = 1, M, 2
    do K = 1, M
        // I = Iu, J = Ju
        C(I,J) = C(I,J) + A(I,K) * B(K,J)
        C(I,J+1) = C(I,J+1) + A(I,K) * B(K,J+1)
        C(I+1,J) = C(I+1,J) + A(I+1,K) * B(K,J)
        C(I+1,J+1) = C(I+1,J+1) + A(I+1,K) * B(K,J+1)
}
```

```
v1.UnrollLoops = {Iu, Ju}
v1.Params = {UI, UJ}
v1.LoopOrder = {K}
v1.Loops = {I,J,K}
v1.WorkingLoops = v1.WorkingLoops - K = {Iu,Ju} // new set to keep loops that have not been mapped yet
```

And
vnew = v1
Vnew = Vnew U vnew = {v1}

Then:
V = {v1}
level = level + 1 = 1
Loops = {I,J,K}
WorkingLoops = WorkingLoops - L = {I,J,K}-{K} = {I,J}

==========
Second iteration of while loop:
==========
V = {v1}
level = 1
Loops = {I,J,K}
WorkingLoops = {I,J}
Refs = {{C(I,J)read, C(I,J)write}, A(I,K), B(K,J)}
v1.Loops = {Iu,Ju,K}; v1.WorkingLoops = {Iu,Ju}; v1.LoopOrder = {K}

Assuming that MostProfitableLoops(WorkingLoops, Refs) returns I, which carries temporal reuse for B(K,J)
(B(K,J) is selected before A(I, J) because it has additional spatial reuse in the innermost loop K):

L = MostProfitableLoops(WorkingLoops, Refs) = I
Vnew = { }
V = {v1}

GenVariant() is called with parameters v1, I, and level==1.

GenVariant: MostProfitableRefs() returns B(K,J), and loops J and K are tiled to generate variants v2 and v2copy:

```
{v2, v2copy} = GenVariant(v1, I, 1)

v2 = {
// control loops have not been ordered yet
do Jc = 1, M, Tj
  do Kc = 1, M, Kt
    do I = 1, M, 2
      do Jt = Jc, Jc+Tj, 2
        do Kt = Kc, Kc+Kt
           // J = Jt, K = Kt
           C(I,J) = C(I,J) + A(I,K) * B(K,J)
           C(I,J+1) = C(I,J+1) + A(I,K) * B(K,J+1)
           C(I+1,J) = C(I+1,J) + A(I+1,K) * B(K,J)
           C(I+1,J+1) = C(I+1,J+1) + A(I+1,K) * B(K,J+1)
}
```

v2copy is v2 with additional code to copy a tile of B(K,J) to a temporary array of size Tk * Tj inserted before loop I.

```
v2copy = {
do Jc = 1, M, Tj
  do Kc = 1, M, Kt
    {copy B(Kc:Kc+Tk, Jc:Jc+Tj+Uj) to temporary array P(1:Tk, 1:Tj+Uj) }
    do I = 1, M, 2
      do Jt = Jc, Jc+Tj, 2
        do Kt = Kc, Kc+Tk
           // J = Jt, K = Kc
           C(I,J) = C(I,J) + A(I,K) * P(K-Kc,J-Jc)
           C(I,J+1) = C(I,J+1) + A(I,K) * P(K-Kc,J-Jc+1)
           C(I+1,J) = C(I+1,J) + A(I+1,K) * P(K-Kc,J-Jc)
           C(I+1,J+1) = C(I+1,J+1) + A(I+1,K) * B(K-Kc,J-Kc+1)
}

v2.WorkingLoops = v2.WorkingLoops - I = {J}
v2.LoopOrder = {Jt,K}
v2.ControlLoops = {Jc,Kc}

Then:
vnew = {v2, v2copy}
Vnew = Vnew U vnew = { } U {v2, v2copy} = {v2, v2copy}  // we lost v1!!!
V = Vnew = {v2, v2copy}
WorkingLoops = WorkingLoops - L = {I,J} - {I} = {J}
level = level + 1 = 2

============================

Third iteration of while loop:

level = 2
```

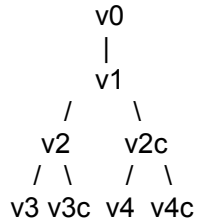
```
WorkingLoops = {J}
Loops = {I,J,K}
V = {v2, v2copy}
v2.WorkingLoops = {J}

MostProfitableLoops returns loop J, which carries temporal reuse for A(I,K).
L = J

Vnew = {}
There are two calls to GenVariant, one for each variant in V:

First call to GenVariant: {v3, v3copy} = GenVariant(v2, J, 2)

MostProfitableRefs returns A(I,K) which has temporal reuse in J;
Loop I is tiled to exploit the reuse carried by J; K is not selected for tiling because
is has already been tiled.

v3 = {
// control loops have not been ordered yet
do Ic = 1, M, Ti
  do Jc = 1, M, Tj
    do Kc = 1, M, Kt
      do It = Ic, Ic+Ti, 2
        do Jt = Jc, Jc+Tj, 2
          do Kt = Kc, Kc+Kt
            // I = It, J = Jt, K = Kt
            C(I,J) = C(I,J) + A(I,K) * B(K,J)
            C(I,J+1) = C(I,J+1) + A(I,K) * B(K,J+1)
            C(I+1,J) = C(I+1,J) + A(I+1,K) * B(K,J)
            C(I+1,J+1) = C(I+1,J+1) + A(I+1,K) * B(K,J+1)
}

v3.WorkingLoops = {J} - L = {J} - {J} = { }
v3.LoopOrder = v3.LoopOrder U {It} = {It, Jt, K}
v3.ControllLoops = {Ic,Jc,Kc}

v3.copy = v3 with additional code to copy a tile of size Ti * Tk to a temporary array
Q, to exploit the reuse of A(Ic:Ic+Ti, Kc:Kc+Tk) in loop Jt

And
vnew = {v3, v3copy}
Vnew = { } U {v3, v3copy} = {v3, v3copy}

====
Second call to GenVariant: {v4, v4copy) = GenVariant(v2copy, J, 2)

And
vnew = {v4, v4copy}
Vnew = Vnew U vnew = {v3, v3copy} U {v4, v4copy} = {v3, v3copy, v4, v4copy}

Then:
V = Vnew = {v3, v3copy, v4, v4copy} // we lost v2, v2copy!
WorkingLoops = { }
```

```
level = level + 1 = 4

}

=====================================================
Notes:

1. With the current algorithm, we don't keep v0, v1, v2 and v2copy in set V; I think we
should revisit this. We could have an additional set V to ke\
ep all variants.

2. Should MostProfitableLoops return a single loop?
```

Sri Hari Krishna Narayanan <snarayan@mcs.anl.gov> Mon, Jan 7, 2013 at 8:59 AM
To: Jacqueline Chame <jacqueline.chame@gmail.com>
Cc: shantharam.manu@gmail.com

Hi Jacque,

Manu and I went through the derivation of variants and found it very useful for our understanding.
We had the following responses to your Notes:
Notes:
1. With the current algorithm, we don't keep v0, v1, v2 and v2copy in set V; I think we should revisit this. We could have an additional set V to keep all variants.
Here is how I see it:



What you are asking is whether there is a utility is keeping the non-leaf nodes around. I would say yes. It probably costs very little to do so. It will be interesting to see if the search algorithm find them useful. This means that a total of 8 variants are provided to the search algorithm.

What Manu and I are curious about is if a variant should be reconsidered for further generation. For example, v0 is not considered as an input to GenVariant after v1 is created. This is because the set newVariants overwrites the set Variants.
So we would like to ask you if you think that the line
Variants <- NewVariants
should instead be
Variants <- Varaints \Union NewVariants

This change would not lead to more iterations of the while loop in DeriveVariants (because that depends on the on the number of Loops and the level),
but could potentially lead to many more calls to GenVariant().

2. Should MostProfitableLoops return a single loop?
At first I think we should settle for one loop, because if we do not, we will have to compare the most profitable reference across multiple loops. I think Manu and I are all keen to have something working now with a possible extension to multiple loops later.

Thanks,
Krishna
[Quoted text hidden]